

# Unified Modeling Language

Dmitriy Andreyev, Akaza Research, Inc.  
August 26, 2003



## UML History

- OOSE (Object-Oriented Software Engineering)  
*Ivar Jacobson*
- The Booch Method  
*Grady Booch*
- OMT (Object Modeling Technique)  
*James Rumbaugh*

---

All based on personal preferences, each method has its own strengths and weaknesses

**1996 - UML (Unified Modeling Language)**



## Why do we need UML?

**UML gives us the tools to design and plan software applications. Its models allow us to design and plan the system by:**

- *finding the ways that the system interacts with its environment*
- *identifying objects and their attributes and methods names*
- *establishing the relationship between objects*



## UML Models

- Use Case Diagrams
- Class Diagrams
- Sequence Diagrams
- Collaboration Diagrams
- Package Diagrams
- Object Diagrams
- State Diagrams
- Activity Diagrams
- Component Diagrams
- Deployment Diagrams



## Use Case Diagrams

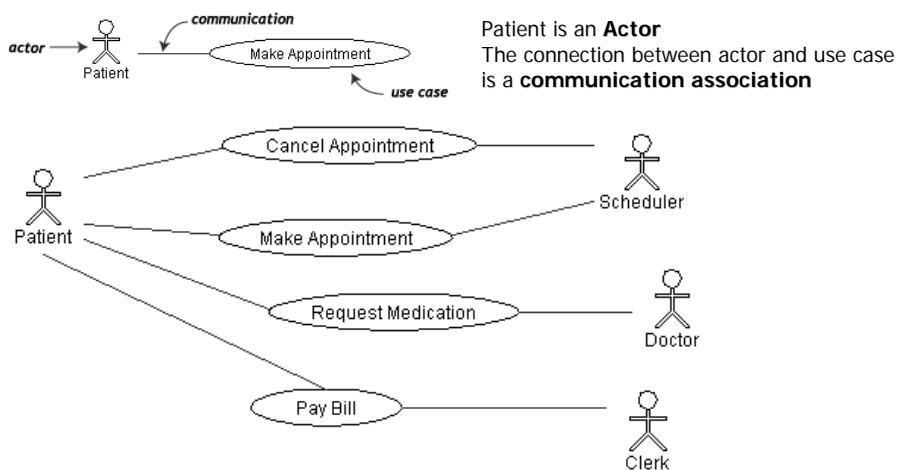
- Show the interaction between the Actor and the various Use Cases
- The Actor is a stick figure, could be another System, initiator of the events
- Each User Case is represented by an oval
- Set relationships with arrows
- Most valuable on large projects and complex systems
- Easy graphical way to display inheritance

### Helpful in three areas:

- Determining features (requirements) – often generate new requirements as the system is analyzed
- Communicating with clients – notational simplicity makes them a good way for developers to communicate with clients
- Generating test cases – can suggest a suite of test cases for collection of scenarios



## Use Case Diagrams



Single Use Case can have multiple Actors.

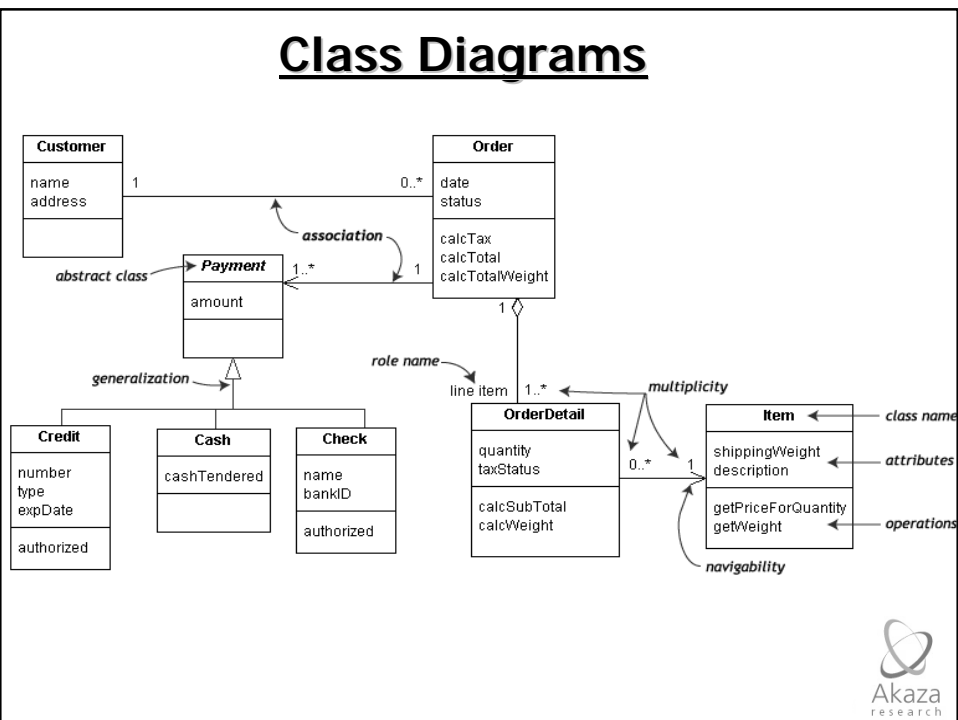


# Class Diagrams

- Describe the various types of objects that exist in the system and the static relationships that exist between them
- One of most useful and most commonly used diagrams
- Allow you to plan how the classes/objects will function and interact
- Classes have properties and methods
  - + indicates public, - indicates private, # indicates protected
  - 'a' indicates abstract class, 'c' indicates concrete class
  - Relationships:
    - Association – between instances of two classes, link that connects 2 classes
    - Derivation (generalization) – inheritance, arrow always points toward parent
    - Composition – filled diamond shape, lives as long as its base class lives
    - Aggregation – an association in which one class belongs to a collection empty diamond shape, can live independently.
    - Multiplicities – 0..\*, 1, 1..\*, 0..1
- Provide a way to easily communicate your ideas to other developers as well as creating documentation of your class structure.



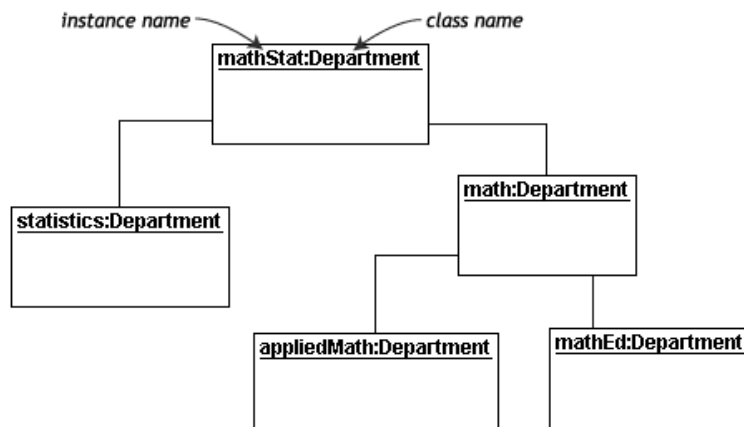
# Class Diagrams



## Object Diagrams

- Show instances instead of classes
- Useful for explaining small pieces with complicated relationships, especially recursive relationships.

## Object Diagrams

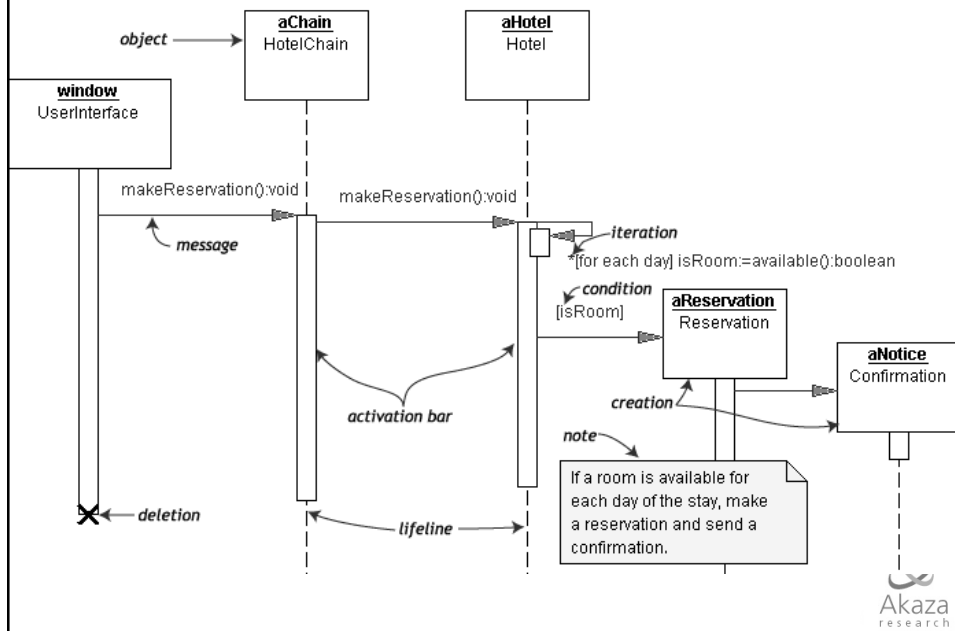


# Sequence Interaction Diagrams

- Interaction diagrams show how groups of objects collaborate in some behavior
- Sequence diagrams are the most common type of Interaction and show an instance of an object and the life of that object
  - Vertical lines show 'lifetime'
  - Big X is terminator
  - Dashed lines show return values
  - Arrows are message calls
  - Expressions in [] are conditions



# Sequence Interaction Diagrams

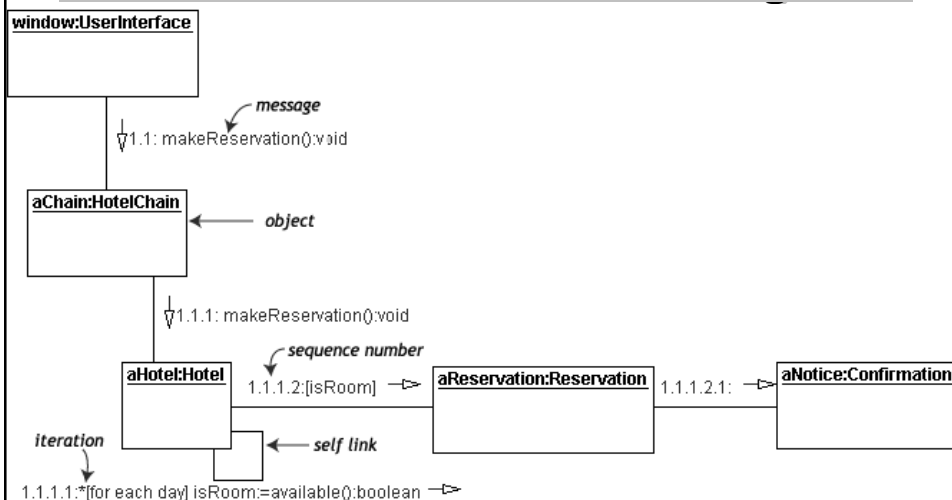


## Collaboration Interaction Diagrams

- In a way similar to Sequence diagram, but focus on object roles instead of times that messages are sent
- Collaboration diagrams show the flow of events and object relations and good when you need to get a quick overview of the general flow of events and object relations
- Objects are shown as icons
- Arrows and numbers indicate the order in which events occur
- Sequence diagrams is a choice when you need to demonstrate the timing and sequencing of events



## Collaboration Interaction Diagrams



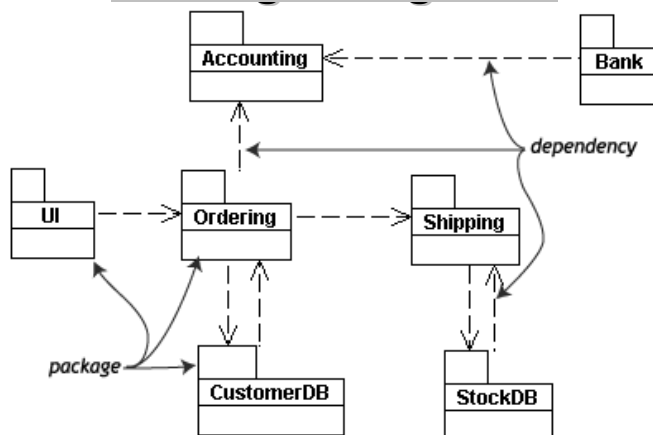
Each message has a sequence number



## Package Diagrams

- Show groups of classes and the dependencies that exist between them.
- Similar to Class Diagrams, just show related classes grouped together
- One package depends on another if changes in the other could possibly force changes in the first
- A dependency exist if any dependency exists between any two classes inside each package
- Useful to get an overview of a large system

## Package Diagrams



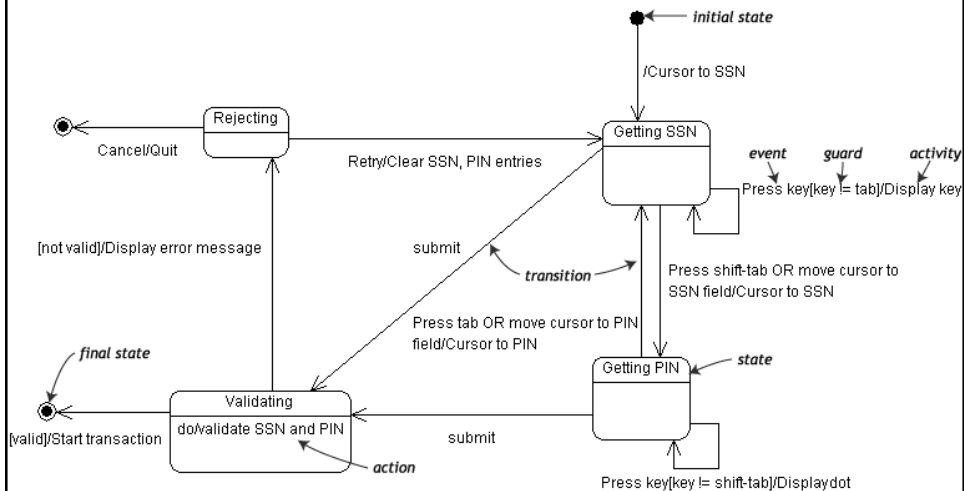
- Package is a rectangle with the small tab on the top.
- Dotted arrows are dependencies.

## State (Statechart) Diagrams

- Describe the behavior of the system
- Show all possible states for each objects and how the state changes as a result of events that happen to it.
- Usually drawn for single class
- States are shown as rounded rectangles
- Transitions are arrows from one state to another
- Initial and final states are dummy states



## State (Statechart) Diagrams

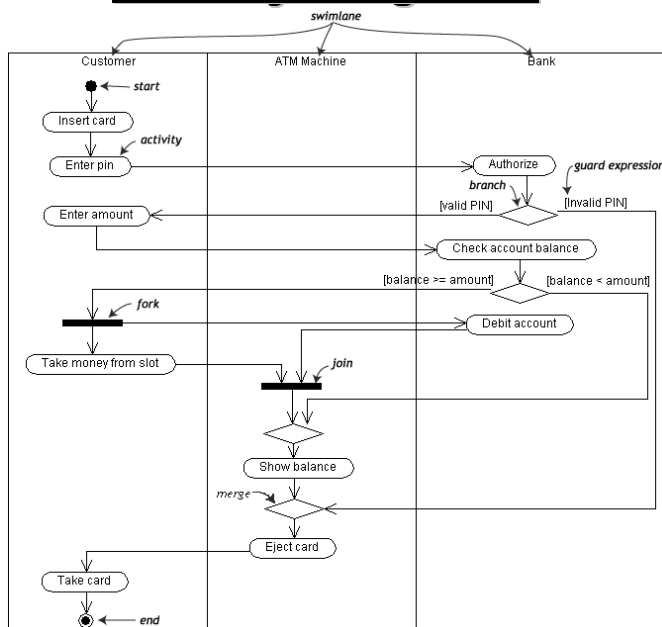


# Activity Diagrams

- Describe the sequencing of activities, show how these activities depend on one another.
- A variant of State Diagrams. It focuses on the flow of activities involved in a single process as opposed to State Diagrams that Focus attention on the object undergoing the process.
- Useful for objects which contain a lot of complex logic that has to be presented clearly.
- Horizontal black lines indicates where the object may take different paths of action.



# Activity Diagrams

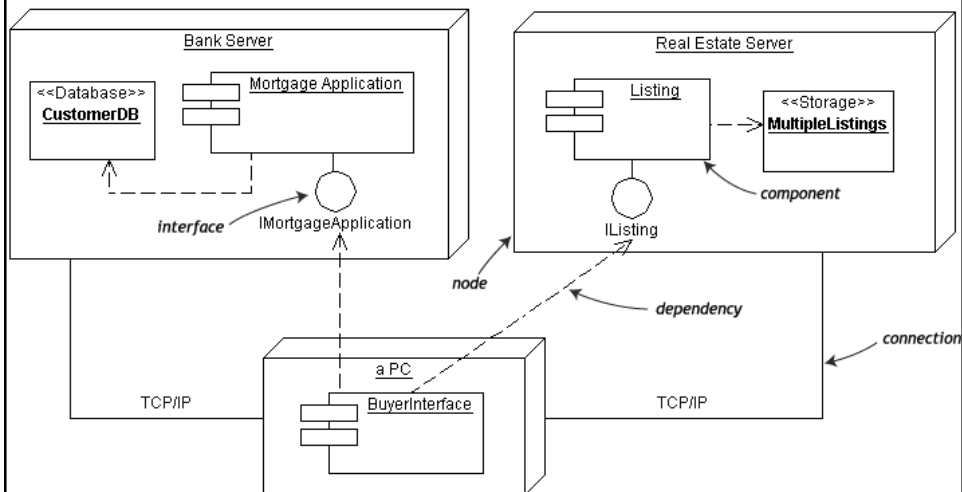


## Component and Deployment Diagrams

- Component Diagrams show the various components of the system and their dependencies, while Deployment Diagrams show how these Components Diagrams interact.
- Usually they combined into a single diagram
- 'Component' represents a physical module of code. They can be different from packages, since they represent the physical packaging of code.
- Dependencies between the components show how changes in one component can affect the other components.



## Component and Deployment Diagrams



Components are shown as rectangles with two tabs at the upper left corner.



## Modeling Tools

- Rational Rose ([www.rational.com](http://www.rational.com)) – well-known, but quite expensive.
- Visual UML ([www.visualobject.com/VisualUMLProducts.htm](http://www.visualobject.com/VisualUMLProducts.htm))
- Microsoft Visio ([www.microsoft.com/office/visio](http://www.microsoft.com/office/visio))

First two allow “Round-Trip” engineering, meaning that you can export the model into the development language of your choice, and the Modeling Wizard will automatically create the classes, properties and methods. Also changes can be “imported” back into the model.

- GDPro ([www.gdpro.com](http://www.gdpro.com))

Modeling tool for Java.



## Summary

Creating UML diagrams can be tedious and time consuming. However, it is necessary part of development process and good planning is an essential component for producing high quality software. Applications that are well-planned take less time consuming to build, more flexible, scalable, easier to maintain as well as easier to accurately estimate the cost of programming the project.

### **Materials used:**

- 1) “An Introduction to Gathering Requirements, Creating Use Cases and UML”, by Ellen Whitney
- 2) “Practical UML. A Hands-On Introduction for Developers”, by Borland.

